Microsoft®
Operating System/2

Programmer's Toolkit

Programmer's Learning Guide

Version 1.0

Microsoft Corporation

Document No. 060060004-100-R00-0388

Contents

Chapter 1
Introduction

1.1 Overview

The *Microsoft® Operating System/2 Programmer's Learning Guide* is intended to help the experienced C programmer make the transition to writing programs that use the Microsoft Operating System/2 application programming interface. The guide provides explanations of how to use the MS® OS/2 functions, types, and data structures to carry out useful tasks, and illustrates these explanations with program samples that you can compile and run with MS OS/2.

1.2 What You Need to Start

To start using this guide, you need the following:

- Experience using MS OS/2
- Experience writing C-language programs

The C programming language is the preferred development language for MS OS/2 programs. Many of the programming features of MS OS/2 were designed with C and other high-level languages in mind. MS OS/2 programs can also be developed in Pascal, FORTRAN, BASIC, and assembly language, but C is the most straightforward and easiest language to use to access MS OS/2 functions. For this reason, all program samples in this guide are written in the C programming language.

Before you start any development, you should review the *Microsoft Operating System/2 Programmer's Reference.* The reference introduces the basic concepts of MS OS/2 and fully defines each MS OS/2 system function. Also, for an explanation of what development tools you need to compile, link, and debug MS OS/2 programs, read the *Microsoft Operating System/2 Programming Tools* manual.

1.3 What This Guide Covers

This guide shows how to use many features of MS OS/2. In particular, it shows how to do the following:

- Use the file-system functions to open, read from, and write to files.
- Use text-based video-input and -output functions to write text to and control the format of the system screen.

- Use the mouse input functions to read events from the mouse, such as mouse motions and button clicks.
- Use the keyboard-input functions to read characters and keystrokes from the system keyboard.
- Use memory-allocation functions to allocate additional memory for your program.
- Use process-control functions to start child processes and threads.

This guide does not cover some of the advanced features of MS OS/2. For example, it does not show how to use monitors, dynamic linking, queues, subsystem registration, or national-language (code-page) support. However, there are several MS OS/2 sample programs provided with the MS OS/2 Programmer's Toolkit that do illustrate these features.

1.4 The Lqh Sample Program

This guide shows how to build a sample program similar to the QuickHelp on-line help program provided with the MS OS/2 Programmer's Toolkit. Chapter 7, "Lqh: A Sample Program," contains an overview of the **lqh** source and an explanation of how the MS OS/2 functions are used in the program.

The **lqh** source files are provided on disk.

1.5 MS OS/2 Sample Programs

In addition to the **lqh** program presented in this guide, there are many additional sample program sources on the disks provided with the MS OS/2 Programmer's Toolkit. These small sample programs demonstrate the features of MS OS/2, including features not described in this guide, such as monitors, dynamic linking, queues, and binding.

The following list briefly describes the MS OS/2 sample programs included in the MS OS/2 Programmer's Toolkit:

Program	Description
alloc	Allocates memory (bound).
argument	Passes parameters to threads.

asmexmpl	Creates and manages threads in assembly language.
asyncio	Shows how to use asynchronous input and output functions to write to the screen.
beepc	Beeps the speaker (bound).
config	Displays the machine configuration (bound).
country	Displays the current country information (bound).
critsec	Uses a critical section to prevent conflict between two threads that are using the **printf** function.
csalias	Creates alias code segments.
cwait	Waits for a child process to exit.
datetime	Prints the date and time (bound).
dosexit	Uses the **DosExit** function to exit threads.
dynlink	Shows how to create C and assembly-language dynamic-link libraries (bound).
exitlist	Sets up an exit routine and then exits.
fsinfo	Gets and sets volume information for drive A (bound).
getenv	Prints the first element of the environment (bound).
hello	Writes "Hello, world" to the screen (bound).
huge	Allocates "huge" memory (bound).
infoseg	Prints information about a process.
iopl	Gets and displays the current cursor position.
keys	Prints keycodes by using the **KbdCharIn** function (bound).
kill	Uses the **DosKillProcess** function to terminate a thread.
machmode	Displays the machine mode (bound).
monitors	Registers and terminates monitors.
move	Moves files (bound).
pipes	Uses pipes.
qhtype	Opens files and prints out handle information (bound).
queues	Uses queues to transfer data between consumer and server processes.
realloc	Increases and decreases the size of a segment.
session	Uses the session-manager functions.

setmaxfh	Sets the maximum number of file handles to 30.
setvec	Modifies the interrupt-vector table (bound).
share	Passes keystrokes between processes that are using shared memory.
signal	Uses a signal handler to catch the CONTROL+C key (bound).
sleep	Sleeps for a while (bound).
suballoc	Uses the suballocation functions (bound).
suspend	Suspends and resumes a thread.
threads	Creates and manages threads.
timer	Sleeps and then beeps three times.
version	Prints the DOS version (bound).
vioreg	Registers a Vio subsystem.

The following sample programs are extended examples that combine many of the features illustrated in the previous samples:

Program	Description
terminal	Emulates a terminal and can be used with a direct connection to a host or through a Hayes modem.
sse	Edits text (bound).
cpgrep	Searches strings, using threads to make the speed of the search as fast as possible.
ds	Displays and moves directory trees, and creates and uses an initialization file in the directory specified by the environment variable USER.
filelist	Lists files in a directory (bound).
life	Demonstrates how to use the the mouse interface in a bound program (bound).
mandel	Displays the Mandelbrot set, using EGA high-resolution graphics.
bigben	Displays a digital clock, demonstrating the Vio functions (bound).
setega	Sets 25/43-line mode (bound).
chaser	Demonstrates the use of threads (requires a mouse).

1.6 MS OS/2 and the C Run-time Library

Many of the sample programs included in the MS OS/2 Programmer's Toolkit combine both MS OS/2 and C run-time functions to carry out their tasks. Although you can use C run-time functions in MS OS/2 programs, the program samples in this guide use MS OS/2 functions exclusively.

In general, there are no special restrictions on using C run-time functions, as long as you use the appropriate version of the C run-time library. For more information about using C run-time functions with MS OS/2 programs, see the *Microsoft C Optimizing Compiler Version 5.1 Update.*

Chapter 2
Overview

2.1 Introduction

This chapter explains the basic steps needed to create an MS OS/2 C-language program. In particular, it explains how to do the following:

- Use the **main** function for your program starting point.
- Use the *os2.h* header file for function declarations.
- Use INCL_ constants to selectively enable function groups.
- Use your program's command line.
- Use MS OS/2 naming conventions in your program sources.
- Use structures in MS OS/2 function calls.
- Use the AND and OR operators to examine and modify bit masks.
- Use care in programs that access shared resources.

2.2 Creating an MS OS/2 Program

Creating an MS OS/2 C-language program is no different than creating any other type of C-language program. You use the **main** function as your program starting point and create and call as many other functions as you need to complete the task. The following simple MS OS/2 program copies the line "Hello, world" to the screen:

```
#include <os2.h>

main( )
{
    USHORT cbWritten;

    DosWrite(1, "Hello, world\r\n", 14, &cbWritten);
}
```

The MS OS/2 system functions use many structures, data types, and constants that are not part of the standard C language. For example, the data type **USHORT** is a special MS OS/2 data type that specifies an unsigned short integer. To access these items, you need to include the MS OS/2 header file *os2.h* at the beginning of your program source file.

The MS OS/2 system functions are not standard C functions. They use the Pascal calling convention. This means, for example, that they expect parameters to be passed in right-to-left order instead of the standard left-to-right order of C functions. Therefore, to use the MS OS/2 functions in a C-language program, you must make sure they are declared with the **pascal** keyword, which directs the C compiler to generate proper

instructions for the function call. Fortunately, all MS OS/2 functions are declared within the *os2.h* file, so including the file saves you the trouble of declaring each function individually.

The *os2.h* file also declares the parameter types for each function. This is convenient since many function parameters would otherwise require type casting to avoid compiler errors. For example, the **DosWrite** function shown in the previous example requires the second parameter to be a full 32-bit (far) address to the given string. Since the *os2.h* file declares the second parameter as such, the cast is carried out for you by the compiler.

2.3 C-Language Header Files

The MS OS/2 C-language header file *os2.h* contains the definitions you need to use the functions, data types, structures, and constants described in the *Microsoft Operating System/2 Programmer's Reference.*

When you include the *os2.h* file, the C preprocessor automatically defines many, but not all, of the most commonly used MS OS/2 functions. The *os2.h* header file is the first file of a set of files that contains the MS OS/2 function definitions. Each file contains definitions for the functions, data types, structures, and constants associated with a specific group of MS OS/2 functions. To minimize the time required to process the many header files, each function group is conditionally processed depending on whether a corresponding constant is defined within the program source file. The following is a list of these constants with descriptions of the function groups they represent:

Constant	Meaning
INCL_BASE	Includes all MS OS/2 1.0 system function definitions.
INCL_DOS	Includes all MS OS/2 1.0 kernel function definitions (Dos).
INCL_SUB	Includes all MS OS/2 1.0 video, keyboard, and mouse functions (Vio, Kbd, and Mou).
INCL_DOSDATETIME	Includes all date/time and timer functions.
INCL_DOSDEVICES	Includes the device and IOPL support functions.
INCL_DOSERRORS	Includes the MS OS/2 error constants.
INCL_DOSFILEMGR	Includes all file-management functions.

INCL_DOSINFOSEG	Includes all information-segment functions.
INCL_DOSMEMMGR	Includes all memory-management functions.
INCL_DOSMODULEMGR	Includes all module-manager functions.
INCL_DOSMONITORS	Includes all monitor functions.
INCL_DOSNLS	Includes national-language-support functions.
INCL_DOSPROCESS	Includes all process- and thread-support functions.
INCL_DOSQUEUES	Includes all queue and other miscellaneous functions.
INCL_DOSRESOURCES	Includes resource-support functions (not available in MS OS/2 1.0).
INCL_DOSSEMAPHORES	Includes all semaphore functions.
INCL_DOSSESMGR	Includes all session-manager functions.
INCL_DOSSIGNALS	Includes all signal functions.
INCL_NOCOMMON	Excludes any function group not explicitly defined.

To use a function within your program function, you simply define the corresponding constant by using the **#define** directive before including the *os2.h* file. For example, the following program includes definitions for the memory-manager and file-system functions:

```
#define INCL_DOSMEMMGR
#define INCL_DOSFILEMGR
#include <os2.h>

main( )
{
   .
   .
   .
}
```

Once you have defined a constant, you may use any function, structure, or data type in that function group.

2.4 A Simple Program: Echoing the Command Line

In standard C-language programs, you can use the *argc* and *argv* parameters of the **main** function to retrieve individual copies of the command-line arguments. You can use these parameters in MS OS/2 programs, but you can also retrieve the entire command line, exactly as the user typed it, by using the **DosGetEnv** function.

When it starts a program, MS OS/2 prepares an environment segment for the program that contains definitions of all environment variables, as well as of the command line. The **DosGetEnv** function retrieves the segment selector for the program's environment segment and the address offset within that segment for the start of the command line.

You can echo the command line on the screen by using the **DosGetEnv** function to get the address of the command line in the environment segment, as shown in the following sample program:

```
#define INCL_DOSQUEUES
#include <os2.h>

main( )
{
    SEL selEnvironment;
    USHORT offCommand;
    PSZ pszCommandLine;
    USHORT cbWritten;
    USHORT i, cch;

    DosGetEnv(&selEnvironment, &offCommand);
    pszCommandLine = MAKEP(selEnvironment, offCommand);

    for (i = 0; pszCommandLine[i]; i++);
    for (i++, cch = 0; pszCommandLine[cch + i]; cch++);

    DosWrite(1, &pszCommandLine[i], cch, &cbWritten);
}
```

The command line is in two parts. The first part is the program name, terminated by a zero byte. The second part is the rest of the command line, terminated by two zero bytes. This sample program echoes the command line by skipping over the program name, then writing everything up to the next zero byte to the screen. The first **for** statement skips over the command name; the second **for** statement computes the length of the string. The **MAKEP** macro creates the far pointer that is needed to access the command line in the environment segment.

You can also examine your program's environment by using the selector retrieved by the **DosGetEnv** function. The program's environment consists of the environment variables that have been declared and passed to the program. Each program has a unique environment that is typically

inherited from the program that started it; for example, from the MS OS/2 command processor **cmd**.

You can use the **DosScanEnv** function to scan for a specific environment variable. This function takes the name of the environment variable that you are interested in and copies its current value to a buffer that you supply. The following program uses **DosScanEnv** to display the value of the environment variable specified in the command line:

```
#define INCL_DOSQUEUES
#include <os2.h>

main( )
{
    SEL selEnvironment;
    USHORT offCommand;
    PSZ pszCommandLine;
    PSZ pszValue;
    USHORT cbWritten;
    USHORT i, cch;

    DosGetEnv(&selEnvironment, &offCommand);
    pszCommandLine = MAKEP(selEnvironment, offCommand);

    for (i = 0; pszCommandLine[i]; i++);
    for (i++; pszCommandLine[i] == ' '; i++);

    if (!DosScanEnv(&pszCommandLine[i], &pszValue)) {
        for (cch = 0; pszValue[cch]; cch++);
        DosWrite(1, pszValue, cch, &cbWritten);
    }
}
```

2.5 Using the MS OS/2 Naming Conventions

The sample programs in this manual use the MS OS/2 naming conventions for their variables and functions. These conventions define how to create names that indicate both the purpose and data type of an item used with the MS OS/2 system functions. When you use the conventions in your source files, you help others who may read your sources to readily identify the purpose and type of the functions, variables, structures, fields, and constants.

The following list briefly describes the MS OS/2 naming conventions:

Item	Convention
Variable	All names consist of three elements: a prefix, a base type, and a qualifier. The base type identifies the data type of the item; the prefix specifies additional information, such as whether the item is a pointer, an array, or a count of

	bytes; and the qualifier specifies the purpose of the item. The prefix and base type are lowercase, and the qualifier is mixed-case.
Parameter	Same as a variable.
Structure field	Same as a variable.
Structure	All names consist of a word or phrase that specifies the purpose of the structure. All letters in the name are uppercase.
Constant	All names consist of a prefix, derived from the name of the function associated with the constant, and a word or phrase that specifies the meaning of the constant in terms of a value, action, color, or condition. All letters in the name are uppercase and an underscore separates the prefix from the rest of the name.
Function name	All names consist of a three-letter system prefix followed by a word or phrase that describes the action of the function. Each word in the function name starts with an uppercase letter. Verb-noun combinations, such as **DosGetDateTime**, are recommended.

The following examples show some of the standard prefix and base types you will see in this manual:

```
/* Base Types */
BOOL fSuccess;      /* f    Boolean flag. TRUE if successful    */
CHAR chChar;        /* ch   8-bit character                     */
SHORT sRate;        /* s    16-bit signed integer               */
LONG lDistance;     /* l    32-bit signed integer               */
UCHAR uchScan;      /* uch  8-bit unsigned character            */
USHORT usHeight;    /* us   16-bit unsigned integer             */
ULONG ulWidth;      /* ul   32-bit unsigned integer             */
BYTE bAttribute;    /* b    8-bit unsigned integer              */
CHAR szName[];      /* sz   zero-terminated array of characters */
BYTE fbMask;        /* fb   array of flags in a byte            */
USHORT fsMask;      /* fs   array of flags in a short           */
ULONG flMask;       /* fl   array of flags in a long            */
SEL selSegment;     /* sel  16-bit segment selector             */

/* Prefixes */
PCH pchBuffer;      /* p    32-bit far pointer to a given type  */
NPCH npchBuffer;    /* np   16-bit near pointer to a given type */
CHAR achData[1];    /* a    array of a given type               */
USHORT ichIndex;    /* i    index to an array of a given type   */
USHORT cb;          /* c    count of items of a given type      */
HFILE hf;           /* hf   handle identifying a given object   */
USHORT offSeg;      /* off  offset                              */
USHORT idSession;   /* id   identifier for a given object       */
```

When you are naming variables, remember that the prefix and base type are optional for common integer types such as **SHORT** and **USHORT**.

2.6 Using Structures: Getting the Time of Day

Many MS OS/2 functions use structures for input and output parameters. To use a structure in an MS OS/2 function, you first define the structure in your program, then pass a 32-bit far address to the structure as a parameter in the function call.

For example, the **DosGetDateTime** function copies the current date and time to a **DATETIME** structure whose address you supply. The fields of the **DATETIME** structure define the month, day, and year, as well as the time of day (to hundredths of a second). The **DATETIME** structure, defined in the *os2.h* file, has the following form:

```
typedef struct _DATETIME {    /* date */
    UCHAR   hours;
    UCHAR   minutes;
    UCHAR   seconds;
    UCHAR   hundredths;
    UCHAR   day;
    UCHAR   month;
    USHORT  year;
    SHORT   timezone;
    UCHAR   weekday;
} DATETIME;
```

To retrieve the date and time, you call the **DosGetDateTime** function and use the address operator (&) to specify the address of your **DATETIME** structure in the call. The following example shows how to make the call:

```
#include <os2.h>

CHAR szDayName[] = "MonTueWedThuFriSatSun";
CHAR szMonthName[] = "JanFebMarAprMayJunJulAugSepOctNovDec";
CHAR szDate[] = "xx:xx:xx xxx xxx xx, xxxx\r\n";

main( )
{
    DATETIME date;
    SHORT offset;
    SHORT i;
    USHORT usYear;
    USHORT cbWritten;

    DosGetDateTime(&date);  /* Address of the DATETIME structure */

    szDate[0] = (date.hours/10) + '0';
    szDate[1] = (date.hours%10) + '0';
    szDate[3] = (date.minutes/10) + '0';
    szDate[4] = (date.minutes%10) + '0';
    szDate[6] = (date.seconds/10) + '0';
    szDate[7] = (date.seconds%10) + '0';
    offset = date.weekday * 3;
    for (i = 0; i < 3; i++)
        szDate[i + 9] = szDayName[i + offset];
```

```
    offset = (date.month - 1) * 3;
    for (i = 0; i < 3; i++)
        szDate[i + 13] = szMonthName[i + offset];
    szDate[17] = (date.day < 10)?' ':(date.day/10 + '0');
    szDate[18] = (date.day%10) + '0';
    usYear = date.year;
    szDate[21] = (usYear/1000) + '0';
    usYear = usYear % 1000;
    szDate[22] = (usYear/100) + '0';
    usYear = usYear % 100;
    szDate[23] = (usYear/10) + '0';
    szDate[24] = (usYear%10) + '0';

    DosWrite(1, szDate, 27, &cbWritten);
}
```

One drawback of using MS OS/2 functions exclusively is that there are no formatted output functions, such as the **printf** function. This sample program, therefore, formats the data itself before displaying it. The program uses the integer-division operators (/ and %) to convert binary numbers to ASCII characters. The program then copies the ASCII characters to a string and displays the string by using the **DosWrite** function.

Some MS OS/2 functions require that you fill one or more fields of the structure before making the function call. For example, there are some structures whose length depends on the version of the operating system being used; MS OS/2 requires that you supply the expected length so that the function does not copy data beyond the end of your structure.

2.7 Using Bit Masks

In MS OS/2, many functions use bit masks. A bit mask (also called an array of flags) is a combination of two or more Boolean flags in a single byte, word, or double-word value. In C-language programs, you can use the bitwise AND, OR, and NOT operators to examine and set the values in a bit mask.

If a function retrieves a bit mask, you can check a specific flag in the bit mask by using the AND operator, as shown in the following example:

```
USHORT fsEvents;

if (fsEvent & 0x0004)
    /* Is the flag set */
```

Or you can set a flag in a bit mask by using the OR operator, as shown in the following example:

```
ULONG flFunctions;

flFunctions = flFunctions | KR_KBDPEEK;
```

Finally, you can clear a flag in a bit mask by using the AND and NOT operators, as shown in the following example:

```
USHORT fsEvent;

fsEvents = fsEvent & ~0x0004;
```

2.8 Sharing Resources: Playing a Tune

Many MS OS/2 functions let you use the resources of the computer, such as the keyboard, screen, disk, and even the system speaker. Since MS OS/2 is a multitasking operating system and more than one program can run at a time, MS OS/2 considers all resources of the computer to be shared resources. As a result, programs must cooperate with other running programs and must not claim exclusive access to a given resource.

Consider a simple program that plays a short tune by using the **DosBeep** function. This function, when called by a single program, generates a tone at the system speaker, but if two programs call **DosBeep** at the same time, the result is chaos. Try running two or more copies of the following program at the same time:

```
#include <os2.h>

#define CNOTES 14
USHORT ausTune[] = {
    440,1000,
    480,1000,
    510,1000,
    550,1000,
    590,1000,
    620,1000,
    660,1000
    };

main( )
{
    int i;

    for (i = 0; i < CNOTES; i+ = 2)
        DosBeep(ausTune[i], ausTune[i + 1]);
}
```

The first argument to the **DosBeep** function specifies the frequency of the note. The second argument specifies the duration. The array ausTune defines values for the frequency and duration of each note in the tune.

DosBeep is intended to be used for signaling the user when an error occurs, such as pressing an incorrect key. Since the system speaker is a shared resource, a process should use the **DosBeep** function sparingly.

Chapter 3
Input and Output

3.1 Introduction

Input and output are two of the most important tasks that any program carries out. This chapter explains how to read from and write to files on disks and other input and output devices, such as printers, modems, and the system console.

3.2 Opening Files

Before carrying out any input or output operation, you need a file handle. A file handle is a 16-bit value that identifies the file or device that you want to read from or write to. You can create a file handle by using the **DosOpen** function, which opens the specified file and returns a file handle for it. For example, in the following statements, **DosOpen** opens the existing file *simple.txt* for reading and copies the file handle to the hf variable:

```
HFILE hf;
USHORT usResult;

DosOpen("simple.txt",    /* filename                    */
    &hf,                 /* file handle                 */
    &usResult,           /* action taken                */
    0,                   /* size                        */
    0,                   /* file attribute              */
    0x0001,              /* open method                 */
    0x0040,              /* access and sharing method   */
    0);                  /* reserved                    */
```

If the **DosOpen** function opens the file, it copies the file handle to the hf variable and copies a value to the usResult variable to indicate what action was taken (for example, 0x0001 for "existing file opened"). To open an existing file, a size and file attribute are not needed, so the fourth and fifth arguments are set to zero. The sixth argument, 0x0001, directs **DosOpen** to open the file if it exists or to return an error if it does not exist. The next argument, 0x0040, directs **DosOpen** to open the file for reading only and let other programs open the file even while the current program has it open. The final argument is reserved and should always be zero.

The **DosOpen** function indicates that it successfully opened the file by returning zero. You can then use the file handle returned in the hf variable in subsequent functions to read data from the file or to check the status or other characteristics of the file. If **DosOpen** fails to open the file, it returns an error value. For example, if the file cannot be found in the current directory, the function returns the value 0x0002.

When you open a file, you must specify whether you want to read from the file, write to it, or both read and write. You must also specify whether you want other processes to have access to the file while you have it open. You do this by combining two values. These values specify the access and sharing methods and are described in the following list:

Value	Meaning
0x0000	Open a file for reading.
0x0001	Open a file for writing.
0x0002	Open a file for reading and writing.
0x0010	Open a file for exclusive use, denying read and write access by other processes.
0x0020	Deny write access to a file by other processes.
0x0030	Deny read access to a file by other processes.
0x0040	Open a file with no sharing restrictions, granting read and write access to all processes.

In general, you may combine any access method (read, write, or read and write) with any sharing method (deny reading, deny writing, deny reading and writing, or grant any access). Some combinations have to be handled carefully, however, such as opening a file for writing without denying access to it by other processes.

3.3 Reading and Writing to Files

Once you have opened a file and have a file handle, you can read from or write to the file by using the **DosRead** or **DosWrite** function. The **DosRead** function copies a specified number of bytes (up to the end of the file) from the file to the buffer you specify. The **DosWrite** function copies bytes from a buffer to the file.

To read from a file, you must open it for reading, or reading and writing. The following example shows how to open the file named *sample.txt* and read the first 512 bytes from it:

```
HFILE hf;
USHORT usResult;
BYTE abBuffer[512];
USHORT cbRead;

if (!DosOpen("sample.txt", &hf, &usResult, 0, 0,
        0x0001, 0x0040, 0)) {
    DosRead(hf, abBuffer, 512, &cbRead);
    DosClose(hf);
}
```

If the file does not have 512 bytes, **DosRead** reads up to the end of the file and copies the number of bytes read to the cbRead variable. If the function has already read to the end of the file and there are no more bytes to read, it copies zero to the cbRead variable.

To write to a file, you must open it first for writing, or for reading and writing. The following example shows how to open the file *sample.txt* again and write 512 bytes to it:

```
HFILE hf;
USHORT usResult;
BYTE abBuffer[512];
USHORT cbWritten;

if (!DosOpen("sample.txt", &hf, &usResult, 0, 0,
        0x0011, 0x0041, 0)) {
    DosWrite(hf, abBuffer, 512, &cbWritten);
    DosClose(hf);
}
```

The **DosWrite** function writes the contents of the buffer to the file. If it fails to write 512 bytes (for example, if the disk is full), the function copies the number of bytes written to the cbWritten variable.

3.4 Creating a File

You can also create new files by using the **DosOpen** function. Once you have created a new file, you may read from or write to it just as you would with an existing file.

To create a new file, set the sixth parameter to the value 0x0010. The **DosOpen** function then creates the file if it does not already exist. In the following example, the **DosOpen** function creates the file *newfile.txt*:

```
HFILE hf;
USHORT usResult;

DosOpen("newfile.txt",   /* filename                              */
    &hf,                 /* file handle                           */
    &usResult,           /* action taken                          */
    0,                   /* size                                  */
    0x0000,              /* normal file attribute                 */
    0x0010,              /* Creates the file if it does not exist */
    0x0011,              /* write access, share with none         */
    0);
```

In this example, **DosOpen** creates the file and opens it for writing. Note that the sharing method denies any access to all processes, so no other process can open the file while it remains open. The new file is empty (it contains no data).

When a new file is created, the attribute argument (fifth) specifies the file attribute. In the preceding example, the attribute argument is 0x0000, so the file is created as a normal file. Other possible file attributes are read-only and hidden, which correspond to the values 0x0001 or 0x0002, respectively.

The file attribute affects how other processes access the file. For example, if the file is read-only, no process may open the file for writing. The one exception to this rule is that the process that creates the read-only file may write to it immediately after creating it. After closing the file, however, the process may not open it for writing again.

When a file is created, the size argument (fourth) specifies the original size of the new file. For example, if 256 is specified, the new file is 256 bytes long. These 256 bytes, however, are undefined. It is up to the program to write valid data to the file. In any case, no matter what size you specify, subsequent calls to the **DosWrite** function copy data to the beginning of the file.

3.5 Closing a File

You can a close a file by using the **DosClose** function. Since each program has a limited number of file handles that it can have open at any given time, it is a good practice to close a file after using it. To do so, just supply the file handle by using the **DosClose** function, as shown in the following example:

```
HFILE hf;
USHORT usResult;
BYTE abBuffer[80];
USHORT cbRead;

if (!DosOpen("sample.txt", &hf, &usResult, 0L,
        0x0000, 0x0001, 0x0040, 0L)) {
    DosRead(hf, abBuffer, 80, &cbRead);
    DosClose(hf);
}
```

If you have opened a file for writing, the **DosClose** function directs the system to flush the file buffer; that is, to write any existing data in the intermediate file buffer to the disk or device. The system keeps intermediate file buffers to make file input and output more efficient. For example, it saves data from previous calls to the **DosWrite** function until a certain number of bytes are in the buffer. It then writes the contents of the buffer to the disk.

3.6 Standard Input and Output Files

Every program, when first starting, has three input and output files available for its use. These files, called the standard input and output files, let the program read input from the keyboard and display output on the screen without requiring you to open or prepare the keyboard or screen.

As the system starts a program, it automatically opens the three standard files and makes the handles of the files—numbered 0, 1, and 2—available to the program. You can then read from and write to the standard files as soon as your program starts.

File handle 0 is the standard input file. This handle lets you read characters from the keyboard by using the **DosRead** function. The function reads the specified number of characters unless the user types a turnaround character; that is, a character that marks the end of a line (the default turnaround character is a carriage-return/newline character pair). As **DosRead** reads the characters, it copies them to the buffer you have supplied, as shown in the following example:

```
BYTE abBuffer[80];
USHORT cbRead;

DosRead(0, abBuffer, 80, &cbRead);
```

In this example, **DosRead** copies the number of characters read from the standard input to the cbRead variable. The function also copies the turnaround character, or characters, to the buffer. If the function reads less than 80 characters, the turnaround character will be the last one in the buffer.

File handle 1 is the standard output file. It lets you write characters on the screen by using the **DosWrite** function. The function writes the characters in the given buffer or string to the current line. If you want to start a new line, you must insert the current turnaround character in the buffer. The following example displays a prompt, reads a string, and displays the string:

```
USHORT cbWritten;
USHORT cbRead;
BYTE abBuffer[80];

DosWrite(1, "Enter a name: ", 14, &cbWritten);
DosRead(0, abBuffer, 80, &cbRead);
DosWrite(1, abBuffer, cbRead, &cbWritten);
```

File handle 2 is the standard error file. It also lets you write characters on the screen. Most programs use this file to display error messages, since this lets the user redirect the standard output to a file without also redirecting error messages to the file.

3.7 Redirecting Standard Files

Although the standard input, output, and error files are usually the keyboard and screen, they are not always. For example, if the user redirects the standard input by using the redirection symbol (>) on the program command line, all data written to the standard output file goes to the given file. For example, the following command line redirects the standard output to the file *sample.txt* and redirects the standard error file to the file *sample.err*:

```
type startup.cmd >sample.txt 2>sample.err
```

When a standard file is redirected, its handle is still available but corresponds to the given disk file instead of to the keyboard or screen. You can still use the **DosRead** and **DosWrite** functions to read from and write to the files.

3.8 Wildcards in Filenames

You cannot use the wildcard characters (* and ?) in filenames that you supply to the **DosOpen** function, but you can locate files that match a given wildcard by using the **DosFindFirst** and **DosFindNext** functions.

The **DosFindFirst** function locates the first file in the current directory that matches a given filename. The filename can contain wildcard characters. The **DosFindNext** function locates the next file that matches, and continues to find additional matches on each subsequent call until all matching names are found. The functions copy the file statistics, such as name, attributes, and creation date, to a structure that you supply.

The following example shows how to find all filenames that have the filename extension *.c*:

```
HDIR hdir;
USHORT usSearch;
FILEFINDBUF findbuf;

usSearch = 1;
hdir = 1;
DosFindFirst("*.c",
    &hdir,              /* directory handle                    */
    0x0000,             /* file attribute to look for          */
    &findbuf,           /* result buffer                       */
    sizeof(findbuf),
    &usSearch,          /* number of matching names to look for */
    0L);
```

```
do {
                        /* Use filename in findbuf.achName      */

    usSearch = 1;
    DosFindNext(hdir, &findbuf, sizeof(findbuf), &usSearch);
} while (usSearch != 0);
DosFindClose(hdir);
```

This example continues to retrieve matching filenames until the **DosFindNext** function returns zero in the usSearch variable. Before each call, the usSearch variable is set to 0x0001 in order to direct the function to look for only one matching name at a time.

To keep track of which files have already been found, both functions use the directory handle hdir to identify the current position in the directory. The directory handle also identifies for **DosFindNext** the name of the file being sought. This handle must be set to 0x0001 or 0xFFFF before the **DosFindFirst** function is called, and the value returned by **DosFindFirst** must be used in subsequent calls to **DosFindNext**.

After locating the files you need, you should use the **DosFindClose** function to close the directory handle. This ensures that when you search for the same files again you will start at the beginning of the file.

3.9 Asynchronous Reading and Writing

The **DosRead** and **DosWrite** functions are synchronous input and output functions, since they carry out their reading and writing operations before returning control to the program. Using asynchronous input and output functions, your program can continue with other tasks while a function reads from or writes to a file in a separate operation. Asynchronous input and output functions minimize the effect of input and output on the speed of your applications.

3.10 Moving the File Pointer

Every disk file has a corresponding file pointer that marks the current location in the file. The current location is the byte in the file that will be read from or written to on the next call to the **DosRead** or **DosWrite** function. Usually, the file pointer is at the beginning of the file when you first open or create it and advances one byte at a time as you read a byte from or write a byte to the file. You can, however, change the position of the file pointer at any time by using the **DosChgFilePtr** function.

The **DosChgFilePtr** function moves the file pointer a specified offset from a given position. You can move the pointer from the beginning of the file, from the end, or from the current position. The following example shows how to move the pointer 256 bytes from the end of the file:

```
HFILE hf;
USHORT usResult;
ULONG ulActual;

DosOpen("fred", &hf, &usResult, 0L, 0, 0x0011, 0x0040, 0L);
DosChgFilePtr(hf, -256, 2, &ulActual);
```

In this example, **DosChgFilePtr** moves the file pointer to the 256th byte from the end of the file (toward the beginning). If the file is not that long, the function moves the pointer to the first byte in the file and returns the actual position (relative to the end of the file) in the ulActual variable.

You can move the file pointer only for disk files. You cannot use **DosChgFilePtr** to change the file pointer's position on the screen, nor can you use it to read ahead from the keyboard.

3.11 Redirecting Standard Files

You can redirect a standard file from within your program by closing the file, then immediately reopening another file to be used for input or output. Whenever you open a file, MS OS/2 uses the lowest available handle for the new file. For example, if you close the standard input file (handle 0) and immediately reopen some other file, that new file receives handle 0. Any subsequent calls to the **DosRead** function that specify handle 0 would be read from the new file, not from the previous standard input.

Redirecting in this way is especially useful if you want to start a child process and have its standard input be a disk file rather than the keyboard. The following example shows how to redirect the standard input file:

```
DosClose(0);
DosOpen("sample.c", &hf, &usResult, 0, 0, 1, 0x0040, 0);
```

3.12 Devices

You can open a number of devices by using the **DosOpen** function. A device is a piece of hardware, other than a disk drive, that is intended to be used for input and output. For example, the keyboard and screen are devices, as are any serial or parallel ports that your computer may have.

MS OS/2 lets you open and access a device just as you would open a disk file. However, what you read from or write to a device depends on the device attached to it. This is not true for disk files. For example, if you open a serial port that has a printer connected to it, you will need to know the input format of the printer. Writing plain text to the printer may or may not give you the result that you want.

The device may also behave differently depending on what driver you have installed to support the device. For example, if you write to the system console, each byte is interpreted as a character and is subsequently displayed on the screen. If, however, you load the ANSI display driver when you start the system, some byte sequences may represent actions to take, such as moving the cursor.

You can open any device by using the **DosOpen** function, but to do so you must supply the special reserved name for that device. For example, to open the console (both keyboard and screen), you must specify the name **con**. MS OS/2 interprets this to be the console device and opens the keyboard and screen devices for reading and writing.

There are several reserved device names. The following is a list of those commonly used in programs:

Device name	Description
con	System console. This device consists of both the keyboard and screen. You can open it for reading, writing, or both reading and writing. If you open this device for reading only, only the keyboard is read. If you open the device for writing only, only the screen is written to.
com1	Serial port 1. This device is the first serial port in your computer. You can open it for reading, writing, or both reading and writing.
prn	Default printer port. This device corresponds to one of the system parallel ports. You may open it for writing but not for reading.
lpt1	Parallel port 1. This device represents a parallel port.
nul	Null device. This device provides a method of discarding output. If you open this device for writing, any data written to the file is discarded. If you open the device for reading, any attempt to read from the file returns an end-of-file mark.
screen$	System screen. This device is the system screen. It can be written to but not read from. Writing to the screen is similar to writing to the system console. Bytes are

displayed as characters unless they represent an escape sequence. Escape sequences direct the screen to carry out actions, such as moving the cursor.

kbd$ Keyboard. This device is the keyboard. It can be read from but not written to. Reading from the keyboard is similar to reading from the console.

The following example shows how to open serial port 1 for reading and writing:

```
DosOpen("com1", &hf, &usResult, 0L, 0x0000, 0x0001, 0x0012, 0L);
```

You may open some devices only if the appropriate driver is already available. For example, you cannot open a serial port unless a communications driver, such as *com01.sys*, has been loaded by using a **device** statement in the system configuration file *config.sys*.

Once you have opened a device, you can use the **DosRead** and **DosWrite** functions to read from and write to the device. After using the device, you should close it by using the **DosClose** function.

3.13 Input and Output Control

Many devices have more than one mode of operation. For example, a serial port typically has a variety of baud rates at which it can operate. Since these modes of operation are unique to the device (that is, they differ from device to device), MS OS/2 does not include specific functions to set or inquire about these modes. Instead, it provides the **DosDevIOCtl** function, which controls device input and output. You can use **DosDevIOCtl** to set and retrieve information about the devices in your computer.

For example, to set the baud rate of serial port 1, you can use the SETBAUDRATE control function (0x0001,0x0041). The following example shows how to set the baud rate:

```
USHORT usBaudRate;
usBaudRate = 9600
DosDevIOCtl(&usBaudRate, 0L, 0x0041, 0x0001, hf);
```

Chapter 4
Keyboard, Mouse, and Screen

4.1 Introduction

This chapter describes how to use the keyboard, mouse, and video functions to read keystrokes, read mouse events, and write characters to the screen. Although you can use the file functions to access the keyboard, mouse, and screen, the keyboard, mouse, and video functions provide a more uniform and direct way to do so.

4.2 Reading Keystrokes

You can read keystrokes from the keyboard at any time by using the **KbdCharIn** function. A keystroke includes not only the character value of the key pressed but also the scan code, the state of the SHIFT keys, and the system time when the key was pressed. **KbdCharIn** is typically used to process keys, such as DIRECTION keys, that the **DosRead** function cannot read, but it can also be used to read any key.

When the user presses a key, MS OS/2 copies the keystroke information, in the form of a **KBDKEYINFO** structure, to the keyboard input buffer. The **KbdCharIn** function removes the keystroke from the input buffer as it copies the information to the specified structure. The following example reads a keystroke from the keyboard:

```
KBDKEYINFO kbciKey;

KbdCharIn(&kbciKey,  /* Copies keystroke info to this structure */
    IO_WAIT,         /* Waits until user presses a key          */
    0);              /* Reads from the physical keyboard        */

if (kbciKey.chChar == 'A')  /* Is it the letter 'A'             */
```

In this example, the function reads from the physical keyboard and waits for a keystroke if none is in the input buffer. The third argument, 0, is the default keyboard handle. It identifies the physical keyboard. Keyboard functions, like file functions, require a handle to identify the keyboard to be accessed. The physical keyboard is always available to programs and does not need to be opened to be used. Keyboard handle 0 and standard input handle 0 are not the same. The standard input file can be closed or redefined, but the handle to the physical keyboard cannot.

You can use the IO_WAIT constant to direct **KbdCharIn** to wait for a keystroke if there are no keystrokes in the buffer. Or you can use the IO_NOWAIT constant to direct the function to return immediately even if there is no keystroke.

4.3 Displaying a Character

The **KbdCharIn** function does not echo a keystroke as it reads it; that is, the function does not write the corresponding character to the screen. If you want to display the character, you can do so by using the function **VioWrtTTy**. The following example writes the letter "A" to the screen:

```
CHAR ch = 'A';

VioWrtTTy(&ch, 1, 0);
```

In this example, the function writes to the system screen. The third argument, 0, is the default video handle, and identifies the system screen. Video functions, like keyboard functions, require a handle to identify the keyboard to be accessed. The system screen is always available to programs and does not need to be opened to be used. Video handle 0 and standard output handle 1 are not the same. The standard output file can be closed or redefined, but the system-screen handle cannot.

If the standard output handle has not been redirected, you can also write a character to the screen by using the **DosWrite** function. **DosWrite** calls the **VioWrtTTy** function to write the character, so when writing to the screen, these functions are identical.

The **VioWrtTTy** function always writes the character to the current position of the cursor, then advances the cursor one position. If the cursor reaches the end of a line, it wraps to the beginning of the next line. If the cursor reaches the end of the screen, **VioWrtTTy** scrolls the screen up by one line.

4.4 Writing a Character to a Specific Location

You can write a character to a specific location on the screen by using the **VioWrtNChar** function. This function lets you specify the row and column on the screen at which to place the character. The following example writes the letter "A" to row 10, column 15:

```
CHAR ch = 'A';

VioWrtNChar(&ch, 1, 10, 15, 0);
```

The **VioWrtNChar** function uses the coordinate system of the screen to determine where to place the character. Such a coordinate system typically divides the screen into rows and columns. Each coordinate represents a character cell, a rectangular area of the screen large enough to display one

character. When you write a character to a particular coordinate, it overwrites the character previously there.

In MS OS/2, the upper-left corner of the screen is at position (0,0) and, for most screen modes, the lower-right corner is at position (24,79). If you attempt to write a character outside of these boundaries, the function returns an error.

You can write repeatedly to the screen by setting the second parameter to a number other than 1. For example, you can clear a 25 × 80 screen by making the following call:

```
CHAR ch = ' ';

VioWrtNChar(&ch, 2000, 0, 0, 0);
```

If it reaches the end of a line, the **VioWrtNChar** function automatically wraps to the beginning of the next line. However, it does not scroll the screen when it reaches the bottom of the screen, and it does not update the cursor position.

4.5 Reading Extended ASCII Keys

Not all keystrokes have corresponding character values. Some keys, such as the DIRECTION keys, generate extended ASCII values. An extended ASCII value is a two-byte value in which the first byte is either zero or 0xE0 and the second byte is the scan code of the key.

To identify a DIRECTION key, you must check both the **chChar** and **chScan** fields of the **KBDKEYINFO** structure. The following sample program searches for DIRECTION keys by reading keystrokes from the keyboard:

```
#define INCL_SUB
#include <os2.h>

main( )
{
    CHAR ch;
    KBDKEYINFO kbciKey;
    USHORT usCol = 40, usRow = 13;

    ch = ' ';
    VioWrtNChar(&ch, 2000, 0, 0, 0);
    ch = 'A';
    do {
        VioWrtNChar(&ch, 1, usRow, usCol, 0);

        KbdCharIn(&kbciKey, IO_WAIT, 0);
```

```
        if (kbciKey.chChar == 0) {  /* Is it extended ASCII? */
            switch (kbciKey.chScan) {
                case 80:                        /* down  */
                     if (usRow < 24) usRow++;
                    break;
                case 72:                        /* up    */
                     if (usRow > 0) usRow--;
                    break;
                case 77:                        /* right */
                    if (usCol < 79) usCol++;
                    break;
                case 75:                        /* left  */
                    if (usCol > 0) usCol--;
                    break;
            }
        }
    } while (kbciKey.chChar != 'q');
}
```

This program updates the usRow and usCol variables by looking for the UP, DOWN, LEFT, and RIGHT DIRECTION keys from the keyboard. Before reading from the keyboard, the example writes the letter "A" to the location specified by the current usRow and usCol values. This means that the user can use the DIRECTION keys to leave a trail of "A's" on the screen. The program continues to loop until the user presses the Q key.

4.6 Using the Mouse

In addition to using the keyboard, you can use the mouse for program input. To use the mouse, you must open it first. MS OS/2 does not provide a default handle for the mouse as it does for the keyboard and screen. Once you have opened the mouse, you may read events from the mouse event queue. The system copies an event to the queue whenever the user moves the mouse, or presses or releases a mouse button.

You can open the mouse by using the **MouOpen** function. This function takes the name of the device driver and a pointer to the variable that receives the mouse handle. The following example opens the mouse by using the default device driver:

```
HMOU hmou;

MouOpen(NULL, &hmou);
   .
   .
   .
MouClose(hmou);
```

In this example, the NULL argument directs **MouOpen** to use the mouse device driver specified in the first **device** command in the *config.sys* file. The function opens the mouse and copies a handle to the variable hmou.

This handle can be used in subsequent mouse functions to carry out tasks, such as reading from the event queue.

As with files in the file system, the mouse is a shared resource, available to every program in the current screen group. If two or more programs open and use the mouse, what one does affects the other. For this reason, you should close the mouse when you no longer need it. You can close the mouse by using the **MouClose** function.

Once you have a mouse handle, you need to show the mouse pointer by using the **MouDrawPtr** function. When the mouse is first opened, the mouse pointer is hidden from view, so even though the user can use the mouse and your program can process mouse input, there is nothing to tell the user where the mouse is located. The following example opens the mouse and then shows the mouse pointer:

```
HMOU hmou;

MouOpen(OL, &hmou);
MouDrawPtr(hmou);
   .
   .
   .
```

If you plan to write to the screen when the mouse pointer is visible, be careful not to write directly over the pointer. The system displays the pointer by combining the mouse pointer-shape masks with the contents of the screen at the current pointer position. When the user moves the mouse, the system restores the previous contents of the screen, destroying whatever was there. This means that if you write to the screen while the pointer is visible, what you write will be lost when the mouse next moves.

If you need to write to a screen position that is occupied by the mouse pointer, you can hide the pointer temporarily by using the function **MouRemovePtr**. This function specifies an exclusion rectangle. The mouse pointer, upon moving into this rectangle, disappears. The following example shows how to hide the mouse pointer when writing to the screen:

```
NOPTRRECT mourtRect;

mourtRect.row = 0;
mourtRect.col = 0;
mourtRect.cRow = 24;
mourtRect.cCol = 79;

MouRemovePtr(&mourtRect, hmou);
VioWrtNChar(&ch, 1, 10, 10, 0);
MouDrawPtr(hmou);
```

In this example, the entire screen is specified as the exclusion rectangle. The **NOPTRRECT** structure takes the coordinates of the upper-left corner of the screen and the width and height of the desired rectangle. The mouse pointer is shown immediately after the character is written by using the **MouDrawPtr** function.

You can use the **MouReadEventQue** function to read from the mouse event queue. This function copies the next event (if any) in the queue to a **MOUEVENTINFO** structure. The structure has four fields: **fs**, **Time**, **row**, and **col**. The **fs** field specifies the action that generated the event; for example, if the mouse moved, the field is set to 0x0001. The **row** and **col** fields specify the location of the mouse when the event occurred, and the **Time** field specifies the system time. If there is no event in the queue when you call the **MouReadEventQue** function, the function fills the structure with zeros. Since zero is a valid value for the **fs**, **row**, and **col** fields, you need to check the **Time** field to see if you received an event (the **Time** field will be nonzero if you received an event).

The following sample program opens the mouse and reads events from the mouse event queue. Each time the user presses the mouse button, the program writes the letter "A" to the location of the mouse pointer on the screen:

```
#define INCL_SUB
#include <os2.h>

NOPTRRECT mourtRect = { 0, 0, 24, 79 };

main( )
{
    CHAR ch;
    KBDKEYINFO kbciKey;
    HMOU hmou;
    MOUEVENTINFO moueiInfo;
    USHORT fWait = 0;

    ch = ' ';
    VioWrtNChar(&ch, 2000, 0, 0, 0);
    ch = 'A';

    MouOpen(0L, &hmou);
    MouDrawPtr(hmou);

    do {
        MouReadEventQue(&moueiInfo, &fWait, hmou);
        if (moueiInfo.Time) {
            if (moueiInfo.fs & 0x0004) { /* Is 1st button down? */
                MouRemovePtr(&mourtRect, hmou);
                VioWrtNChar(&ch, 1, moueiInfo.row, moueiInfo.col, 0);
                MouDrawPtr(hmou);
            }
        }
        KbdCharIn(&kbciKey, IO_NOWAIT, 0);
    } while (kbciKey.chChar != 'Q');

    MouClose(hmou);
}
```

After retrieving an event from the queue (and verifying that it is a valid event), this sample program checks the **fs** field to see if it contains the value 0x0004. If it does, then the first mouse button (typically the leftmost button) is down. The program then hides the pointer and writes the letter "A" to the screen. After restoring the mouse pointer, the program checks

the keyboard to determine whether the user has pressed the Q key. The program continues to read events from the queue until the user presses the Q key.

4.7 Selecting the Events to be Queued

You may have noticed that the program described in the previous section seemed to ignore some mouse events if you moved the mouse quickly. The program was not ignoring the events, it just was not receiving them. The mouse event queue is small, and once it is full, any new event bumps the oldest event from the queue. This means that you may lose information if your program cannot read from the queue fast enough.

One way to minimize the amount of information lost is to exclude certain events from being placed in the queue. You can do this by using the **MouSetEventMask** function. This function lets you disable the mouse events that you do not wish to process. For example, in the previous program, you do not need the mouse-motion events, so disabling them would reduce the chances of losing a button-down event.

You disable an event by setting the bits in the event mask for only those events that you want to process. The following example enables the first button-down event, but disables all other events:

```
USHORT fsEvents;

fsEvents = 0x0004;

MouSetEventMask(&fsEvents, hmou);
```

If you disable an event such as a button press, the system will do nothing to the queue if the user presses that button, but it will internally record that button press. If some other event occurs while the button is still down, the system adds this information to the event that it passes to the queue. In other words, if you have disabled button-down events, the **fs** field in the **MOUEVENTINFO** structure will still indicate when that button is down.

Since most mouse events correspond to the press or release of a mouse button, you can retrieve the number of buttons on the mouse by using the **MouGetNumButtons** function. This can help you decide which events to enable and which to disable.

You can use the **MouGetNumQueEl** function to retrieve the number of events in the queue. This function retrieves the size of the queue as well as a count of the events in the queue. If you decide that you do not need the events already in the queue, you can flush them from the queue by using the **MouFlushQue** function.

4.8 Reading a String of Characters from the Keyboard

You can read a string of characters from the keyboard by using the **KbdStringIn** function. Using this function is similar to reading input from the keyboard by using the **DosRead** function and the standard input file.

The **KbdStringIn** function, unlike **KbdCharIn**, echoes characters as you type them, so there is no need to write the characters separately. The function reads the specified number of characters or reads up to the end-of-line (or turnaround) character. The following example reads a line of text:

```
CHAR achBuf[80]
STRINGINBUF kbsiLength;

kbsiLength.cb = 80;

KbdStringIn(achBuf, &kbsiLength, IO_WAIT, 0);
```

The **KbdStringIn** function copies the number of characters read to the **cchIn** field in the **STRINGINBUF** structure. You can use this field to enable or disable the MS OS/2 editing keys for your program. These editing keys let the user recall and modify the previously typed line. If you set the **cchIn** field to zero before making the next call to **KbdStringIn**, you disable the editing keys. Otherwise, you enable editing up to the number of characters that you specify.

4.9 Writing a String of Characters to the Screen

You can write a string of characters directly to the screen by using the **VioWrtCharStr** function. When you write characters to the screen in this way, you can specify the row and column at which the characters are to start. This function is similar to the **VioWrtNChar** function, except that, instead of a single character, you specify an array of characters. The following example writes the string "Hello, world" to the middle of the screen:

```
VioWrtCharStr("Hello, world",  12, 13, 34, 0 );
```

4.10 Writing Character Cells to the Screen

You can write character cells to the screen by using the **VioWrtNCell** or **VioWrtCellStr** function. A character cell is a two-byte value that specifies a character and its attribute. A character attribute defines the color, intensity, and appearance of the character to be written. The following example writes a red letter "A" to the middle of a color screen:

```
BYTE bCell[2] = { 'A', 0x04 };

VioWrtNCell(&bCell, 1, 13, 40, 0);
```

Character cells are useful in programs that take full advantage of the text-mode capabilities of their display adapter. However, the meaning and range of values for attributes depend on the device, so it is important that you check the display-adapter type. You can do this by using the **VioGetConfig** function, as shown in the following example:

```
VIOCONFIGINFO viociConfig;

VioGetConfig(0L, &viociConfig, 0);
switch (viociConfig.adapter) {
    case 0:             /* monochrome adapter              */
        break;
    case 1:             /* color graphics adapter (CGA)    */
        break;
    case 2:             /* enhanced graphics adapter (EGA) */
        break;
}
```

4.11 Moving and Hiding the Cursor

If you choose to use the **VioWrtTTy** or **DosWrite** function to write text to the screen, you can control the placement of that text on the screen by using the **VioSetCurPos** and **VioGetCurPos** functions to set and get the position of the cursor. The cursor is the flashing underscore or block on the screen that marks the location that will receive the next character written to the screen. The following example moves the cursor to the middle of the screen and there writes the string "Hello, world":

```
VioSetCurPos(13, 34, 0);
VioWrtTTy("Hello, world", 12, 0)
```

If you choose not to use the cursor, you can remove it from the screen by using the **VioSetCurType** function, which takes a **VIOCURSORINFO** structure. If you set the **attr** field to 0xFFFF, the function hides the cursor. The following example hides the cursor:

```
VIOCURSORINFO viociInfo;

VioGetCurType(&viociInfo, 0);
viociInfo.attr = 0xFFFF;
VioSetCurType(&viociInfo, 0);
```

You can restore the cursor by setting the field to its original value. You can retrieve the original value by using the the **VioGetCurType** function.

In the previous example, the **VioGetCurType** function was used to fill the **VIOCURSORINFO** structure with the current information before the structure was used to hide the cursor. In general, whenever you use an MS OS/2 function that takes values from a structure, you should make sure that all fields contain valid values. In this case, the way to ensure valid values is to fill the structure first by using the **VioGetCurType** function.

You can also use the **VioSetCurType** function to change the shape of the cursor. For example, you can change the shape from an underscore to a block by setting the **yStart** and **cEnd** fields to appropriate values. The following example creates a block cursor:

```
VioGetCurType(&viociInfo, 0);
viociInfo.yStart = 10;
viociInfo.cEnd = 0;
VioSetCurType(&viociInfo, 0);
```

4.12 Reading Characters from the Screen

You can read characters from the screen by using the functions **VioReadCellStr** and **VioReadCharStr**. Reading characters is an easy way to determine the content of the screen. Some programs use this as a method of input. For example, the **lqh** program, described in Chapter 7, "Lqh: A Sample MS OS/2 Program," checks the location of the cursor and reads the line at that point to provide context-sensitive help.

The following example reads a character string at the current cursor position:

```
CHAR achBuffer[80];
USHORT cchBuffer = 80;
USHORT usRow, usCol;

VioGetCurPos(&usRow, &usCol, 0);
VioReadCharStr(achBuffer, &cchBuffer, usRow, usCol, 0 );
```

4.13 Scrolling the Screen

You can scroll all or part of the screen by using the **VioScrollDn**, **VioScrollUp**, **VioScrollLf**, and **VioScrollRt** functions.

The following function scrolls the screen up three lines, leaving three blank lines at the bottom of the screen:

```
BYTE bCell[2] = { ' ', 0x07 };

VioScrollUp(3, 0, 24, 79, 3, bCell,  0);
```

You can also use the scroll functions to clear the screen. Whenever the rectangle that you specify has the same dimension as the screen, the entire screen is cleared. For example:

```
BYTE bCell[2] = { ' ', 0x07 };

VioScrollUp(0, 0, 24, 79, 0xFFFF, bCell, 0);
```

4.14 Using the ANSI Display Mode

You can set the video display to ANSI mode by using the **VioSetAnsi** function. When in ANSI mode, the video display checks for and carries out the action specified by any ANSI escape sequences that are written to the screen by using functions like **VioWrtTTy**. An ANSI escape sequence is a combination of characters, starting with the escape character (27), that specifies a particular action to be taken by the video display, such as moving the cursor or displaying subsequent characters in a new display mode.

For a complete listing of the available ANSI escape sequences, see the *Microsoft Operating System/2 Programmer's Reference.*

4.15 Opening and Using Logical Keyboards

The keyboard, like the mouse, is a shared resource to which all programs in a screen group have access. To avoid conflicts between programs sharing the keyboard, MS OS/2 lets programs open and use logical keyboards. A logical keyboard is like a file in that it has a handle and corresponds to a physical device, the physical keyboard. A program cannot receive input from a logical keyboard (as it can from a file) unless it has requested and received the keyboard focus. Since only one logical keyboard in a screen

group can have the focus at any given time, this is an effective way to manage the keyboard access of all programs in the screen group.

A typical use of a logical keyboard is in a program that has more than one thread that is reading keystrokes. If each thread creates a logical keyboard and then waits for the focus before reading keystrokes, this ensures that one thread does not read keystrokes that are intended for another thread. Consider an editing program that offers multiple windows through which you can edit a file or files. If each window has a separate logical keyboard, then character input from the keyboard intended for one window would never be inadvertently read by another.

You can open a logical keyboard by using the **KbdOpen** function. Once opened, a logical keyboard receives keystrokes only if it has the keyboard focus. You can retrieve the focus for a logical keyboard by using the **KbdGetFocus** function. This function retrieves the focus only if no other logical keyboard has it. Otherwise, it will wait for the focus to be freed if you request the function to wait for the focus.

The following example opens a logical keyboard and requests the keyboard focus:

```
HKBD hkbd;

KbdOpen(&hkbd);

KbdGetFocus(IO_WAIT, hkbd);
   .
   .
   .
/* read from the keyboard */
   .
   .
   .
KbdFreeFocus (HKBD);
```

Once a logical keyboard has the focus, it keeps it until you free the focus by using the **KbdFreeFocus** function, even if another thread calls the **KbdGetFocus** function. This means that you need to be careful to free the focus when you no longer need it. When you have finished reading from the keyboard, you should free the keyboard focus by using the **KbdFreeFocus** function. Even if you intend to read from the logical keyboard again, you must free the focus in order to permit other programs to access it. Your logical keyboard will remain open even if you do not have the focus, and you can request the focus again when you need to read from the keyboard again.

If you have completely finished reading from the keyboard or are about to terminate the program, you can use the **KbdClose** function to close the logical keyboard.

Even though your program may use logical keyboards, the physical keyboard is always available. That is, even if a logical keyboard has the focus, you can still read keystrokes from the physical keyboard by using keyboard handle 0.

4.16 Flushing the Keyboard Buffer

You can flush unwanted keystrokes from the keyboard buffer by using the **KbdFlushBuffer** function. Flushing the buffer causes all existing keystrokes in the buffer to be removed. You would typically flush the physical keyboard's buffer if you did not want to process keystrokes that were carried over from a previous program.

4.17 Setting the Keyboard Input Mode

You can set the keyboard input mode by using the **KbdSetStatus** function. The keyboard input mode defines whether the MS OS/2 control and editing keys are interpreted as special keys or as keystrokes only. The keyboard input mode can be raw or cooked. If it is raw, then only the key combination CONTROL+BREAK is recognized by the system as a special key. All other keys are read as keystrokes. If the input mode is cooked, the system recognizes the keys.

You can set the keyboard mode by first retrieving the current keyboard status in a **KBDINFO** structure, then setting the raw-mode constant in the **fsMask** field. The following example sets the keyboard to raw mode:

```
KBDINFO kbstStatus;

KbdGetStatus(&kbstStatus, 0);
kbstStatus.fsMask = (kbstStatus.fsMask & ~0x0008) | 0x0004;
KbdSetStatus(&kbstStatus, 0);
```

This example makes no assumptions about the keyboard status. It clears the cooked-mode constant, then sets the raw-mode constant.

Chapter 5
Memory Management

5.1 Introduction

This chapter describes how to use the MS OS/2 memory manager to allocate additional memory for your programs. In MS OS/2, system memory is carefully managed by the virtual-memory and memory-protection capabilities of the Intel 80286 microprocessor. MS OS/2 grants your program access to memory only if the memory has been explicitly allocated or made available for your program's use.

In MS OS/2, you allocate memory segments. A segment is one or more bytes of memory (up to 64K bytes). Each segment is identified by a unique value, called a segment selector. A segment selector, combined with a segment offset, yields the address of a byte in the segment.

MS OS/2 lets you allocate any number of segments and then use these segments as additional storage for your program. When MS OS/2 first loads your program, it gives your program at least one data segment, called the automatic segment. This segment contains the global and static data declared in the program. It may also contain the program stack. In any case, this segment has a set size that cannot be changed. MS OS/2 also gives your program one or more code segments, which contain the program code. Like the automatic data segment, their size cannot be changed, but they are protected from any changes, intentional or otherwise.

5.2 Allocating and Using Segments

You can allocate a segment of memory by using the **DosAllocSeg** function. You simply specify the amount of memory that you need and a variable to receive the selector created by **DosAllocSeg** to identify the new segment. The following example allocates 512 bytes of memory, then fills the memory with zeros:

```
SEL selArray;
PCH pchArray;
USHORT i;

DosAllocSeg(512, &selArray, 0);
pchArray = MAKEP(selArray, 0);
for (i = 0; i < 512; i++)
    pchArray[i] = 0;
```

In this example, the selArray variable receives the segment selector created by the **DosAllocSeg** function. To access the bytes in the segment, you need to create a far pointer to the segment. You can do this by using the **MAKEP** macro, which combines the selector and an address offset to create the pointer. In this example, the pointer points to the first byte in the segment. The **for** statement assigns zero to each byte in the segment.

When you have finished using a segment, you can free it by using the **DosFreeSeg** function. Freeing the segment returns that memory to the system's pool of available memory and invalidates the selector that identifies the segment. The data in the freed segment is lost and any attempt to access the segment by using the selector of the freed segment generates a general-protection fault.

The following sample program uses an allocated segment to store a string from the standard input file. The program allocates a 512-byte segment to hold input from the standard input file, then frees the segment after the input has been received and processed:

```
SEL selArray;
PCH pchArray;
USHORT i;
USHORT cbRead;
USHORT cbWritten;

DosAllocSeg(512, &selArray, 0);
pchArray = MAKEP(selArray, 0);
DosRead(0, pchArray, 512, &cbRead);
for (i = 0; i < ; i--)
    if (pchArray[i] >= 'a' && pchArray[i] <= 'z')

DosWrite(1, pchArray, cbRead, &cbWritten);
DosFreeSeg(selArray);
```

5.3 General-Protection Faults and Segment Violations

MS OS/2 operates in the 80286 protected mode. This means that the CPU monitors each program's access to memory and prevents programs from accessing memory that is not explicitly assigned to them. Memory consists of segments. Each segment has a unique selector that identifies it and grants the program access. The segment has a size (in bytes). If a program attempts to access a segment that is not assigned to it, or uses an unknown selector, or attempts to access bytes outside of a segment, the system generates a protection fault. A protection fault is a CPU exception that interrupts normal processing and executes a corresponding exception-handling routine. This routine determines the cause of the exception and either fixes the problem or displays an error message to the user.

General protection faults and segment violations are not recoverable. The system displays the error message and terminates the program. This means that you should be careful about using address offsets and selectors and ensure that they are valid.

5.4 Reallocating a Segment

If a segment that you have allocated is too small or too large to meet your needs, you will avoid general-protection faults or wasted space if you reallocate the segment by using the **DosReallocSeg** function. This function adjusts the size of the segment to a given size without changing the segment selector. In the following example, the **DosReallocSeg** function expands the segment from 512 bytes to 1024:

```
SEL selBuf;

DosAllocSeg(512, &selBuf, 0);
   .
   .
   .
DosReallocSeg(1024, selBuf);
```

If you decrease the size of a segment, the data beyond the new end of the segment is lost. If you increase the segment size, be aware that the function does not fill new memory with zeros.

5.4.1 Checking Available Memory

You can use the **DosMemAvail** function to determine the size of the largest available block of free memory; for example:

```
ULONG ulLargestFreeBlock;

DosMemAvail(&ulLargestFreeBlock);
```

Although **DosMemAvail** is useful, if there are many programs running that allocate memory, the retrieved value may not always be up-to-date.

5.4.2 Allocating Huge-Memory Blocks

You can allocate more memory than 64 kilobytes of memory at a time by using the **DosAllocHuge** function. This function allocates as much memory as is requested, up to the limit available in the system. It allocates several 64-kilobyte segments, but ensures that the segment selectors are consecutive (this does not mean that the segments are in consecutive memory). You can then access the memory by computing the appropriate segment selector and offset with the help of the **DosGetHugeShift** function.

The following example shows how to allocate a huge-memory block and compute the selector offset:

```
SEL selHuge;
PCH pch;
USHORT usShiftCount;
USHORT offSelector;
long l;

DosAllocHuge(5, /* Allocates five 64-kilobyte segments     */
    512,        /* plus 512 additional bytes               */
    &selHuge,   /* first segment selector                  */
    6,          /* Reserves six selectors for reallocation */
    0);         /* allocation flags                        */

DosGetHugeShift(&usShiftCount);
offSelector = 2 << usShiftCount;

pch = MAKEP(selHuge + offSelector*2, 0);

pch[0] = 0;
```

After allocating the huge block, this example creates a pointer to the first byte in the third segment. It then sets the byte to zero.

5.5 Moving and Swapping

Although a program cannot control moving and swapping, the system does move data and code segments and swaps data segments whenever necessary. Moving segments gives the system the opportunity to collect all free space into one contiguous block so that it can offer much larger blocks of memory than is otherwise possible.

Swapping segments occurs whenever a program requests more memory than is free, allowing the system to handle more segments than can fit in physical memory. If there is enough space in the system swap file *swapper.dat*, the system copies one or more data segments to the file. It copies these segments back into memory when they are needed again, possibly swapping other data segments out in the process.

The system may also discard segments if it needs additional space. In a sense, discarding is similar to swapping, except that no copy of the discarded segment is copied to the disk. The system will discard code segments if the space is needed and if the segments were loaded originally from an executable file on a hard disk. In MS OS/2, all code segments are pure—that is, not subject to change during execution—so the system can always retrieve a fresh copy of a discarded code segment from the original executable file. (For programs loaded from floppy disks, the original disk may not be present when needed, so no code segments are discarded.)

The user can specify whether the system moves and swaps segments by using the **memman** command in the *config.sys* file. For special-purpose programs, you may request the user to disable swapping and/or moving. Typically, disabling these features enhances the performance of the system when it is needed for dedicated, time-critical tasks.

5.5.1 Creating and Using Discardable Segments

You can create and use discardable segments by using the **DosAllocSeg** function and the SEG_DISCARDABLE constant. A discardable segment is a convenient way to store data that you need infrequently and can regenerate easily. When you use a discardable segment, the system is free to automatically discard your segment if it needs the space. Since the segment selector of a discarded segment remains valid, you can restore the segment simply by reallocating it to the original size and filling it with the data.

To create a discardable segment, use the SEG_DISCARDABLE constant with the **DosAllocSeg** function as shown in the following example:

```
SEL selTempData;

DosAllocSeg(256, &selTempData, SEG_DISCARDABLE);
```

To access a discardable segment, you must lock it first by using the **DosLockSeg** function. While the segment is locked, you can examine and modify the contents of the segment without danger of the system discarding the segment. After examining or modifying the segment, you can unlock it by using the **DosUnlockSeg** function. Unlocking the segment does not automatically discard the segment, so you can usually access the data again by calling the **DosLockSeg** function. The following example locks the selTempData segment, modifies the data, and unlocks the segment:

```
SEL selTempData;
PCH pchTempData;
int i;

DosLockSeg(selTempData);

pchTempData = MAKEP(selTempData, 0);
for (i = 0; i < 256; i++)
    pchTempData[i] = i;
DosUnlockSeg(selTempData);
```

The system discards the segment when it needs to but only if the segment is not locked. After the segment is discarded, any data in the segment is lost, but the segment still exists. If the program that owns the segment attempts to access the data without reallocating the segment, a protection fault will result.

You must check the segment first to see if it has been discarded. If it has, you can reallocate the segment (to restore it to its original size) and then fill it again. You can check a segment to see if it is discardable by checking the return value of the **DosLockSeg** function. If the value is nonzero, the segment has been discarded. The following example shows how to check for a discarded segment:

```
SEL selTempData;
PCH pchTempData;
int i;

if (DosLockSeg(selTempData)) {
    DosReallocSeg(256, selTempData);
    pchTempData = MAKEP(selTempData, 0);
    for (i = 0; i < 256; i++)
        pchTempData[i] = i;
}
else
    pchTempData = MAKEP(selTempData, 0);
DosUnlockSeg(selTempData);
```

Chapter 6
Processes and Threads

6.1 Introduction

This chapter explains how to use multitasking in your programs. Multitasking is a special feature of MS OS/2 that lets more than one program run at the same time. With multitasking, you can have your program start other programs to do work for it, or have different parts of your program run simultaneously.

To work successfully with multitasking, you need to understand two important terms: process and thread. A process is simply the code, data, and other resources of a program in memory, such as the open files, allocated memory, and so on. MS OS/2 considers every program that it loads to be a process. A thread, which is everything else required to execute the program, consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler.

Every process has at least one thread, and the program executes when the thread is given execution control by the system scheduler. The system scheduler determines which threads should run and when they should run, based on a priority scheme, and determines when each thread last had execution control. Threads of lower priority may have to wait while threads of higher priority complete their tasks.

6.1.1 Starting a Process

You can start a process by using the **DosExecPgm** function. The process you start is called a child process. It is a "child" of the starting, or parent, process and inherits many of the resources owned by the parent process, such as open files.

The following example starts a program named **abc**:

```
CHAR achModuleName[128];
RESULTCODES rescResult;

DosExecPgm(achModuleName,
    128,
    EXEC_SYNC,
    NULL,
    NULL,
    &rescResult,
    abc.exe");
```

This example starts **abc** so that it runs synchronously (as specified by the EXEC_SYNC constant). This means that the parent process temporarily stops while the child process executes, and does not continue until the child process ends.

The MS OS/2 command processor *cmd.exe* uses the EXEC_SYNC constant to start most child processes. That is, the processor waits for the process to end before it continues to prompt for the next command. The command processor also lets you start asynchronous programs by using the **detach** command. When you detach a program, the command processor places it in the background and continues to prompt for input.

6.1.2 Setting the Environment and Program Command Line

When you start a process, it inherits the resources of the parent. This includes open files, such as the standard input and output files. A child process also inherits the resources of the screen group, such as the mouse and video modes, and the environment variables of the parent process.

The **DosExecPgm** function determines which environment and command line that the child process receives. The fourth and fifth arguments are pointers to the command line and environment, respectively. If these pointers are NULL, the child process receives nothing for a command line and only an exact duplicate of the parent process's environment. The parent process can modify this by creating a string (that ends with two null characters) and passing the address of the string to the function, as in the following example, in which the string "test –options" is passed to the child process as its command line:

```
CHAR achModuleName[128];
RESULTCODES rescResult;
CHAR chCommandLine[] = "test -options\0";

DosExecPgm(achModuleName,
    128,
    EXEC_SYNC,
    chCommandLine,
    NULL,
    &rescResult,
   "abc.exe");
```

In general, any number of null-terminated strings can be passed as long as the last string ends with two null characters.

6.2 Running an Asynchronous Child Process

You can start a child process and let it run without having to wait for it to end by using the EXEC_ASYNC constant with the **DosExecPgm** function. If you start a process this way, the function copies the process identifier of the child process to the **codeTerminate** field of the structure **RESULTCODES**. You can use this process identifier to check the progress of the child process or to terminate the process.

By using the EXEC_ASYNCRESULT constant, you can also run a child process asynchronously. This constant directs the system to save a copy of the child process's termination status when the child process terminates. This status specifies the reason that the child process stopped. The parent process can retrieve the termination status by using the **DosCWait** function.

6.3 Waiting for a Child Process to End

You can synchronize the execution of a process with one of its child processes by using the **DosCWait** function. When a process calls the function, the process waits until the specified child process has finished before returning. This can be useful, for example, if the parent process needs to ensure that the child process has completed its task before the parent process can continue with its own task.

In the following example, the parent process waits for the child process that is specified by the process identifier in the variable pidChild:

```
RESULTCODES rescResult;
PID pidParent;
PID pidChild;

DosCWait(DCWA_PROCESS, DCWW_WAIT,
    &rescResult, &pidParent, pidChild);
```

You can wait for all child processes to end by using the constant DCWA_PROCESSTREE with the **DosCWait** function.

6.4 Retrieving the Termination Status of a Child Process

You can retrieve the termination status of the most recently terminated process by using the DCWW_NOWAIT constant with the **DosCWait** function. This constant directs the function to return immediately without waiting for a process to end. Instead, it retrieves the termination status from the most recent process to end. MS OS/2 saves the termination status for a process if the process was started by using the constant EXEC_ASYNCRESULT.

6.5 Ending a Process

You can exit a process by using the **DosExit** function. When you exit a process, the system stops the process and frees any existing resources that are owned by it. If no other invocation of the process is running, the system frees the code and data segments of the process.

In the following example, the **DosExit** function is used to exit the process if the given file does not exist:

```
main( )
{
    HFILE hf;
    USHORT usAction, cbWritten;

    if (DosOpen("sample.txt", &hf, &usAction,
            OL, 0, 0x42, 0x1, OL)) {
        DosWrite(2, "Cannot open file\r\n", 18, &cbWritten);
        DosExit(EXIT_PROCESS, 1);
    }

}
```

The EXIT_PROCESS constant directs the function to exit not just the process, but the thread that is calling the function, as well. The **DosExit** function includes an error code that is returned to the parent process through the **RESULTCODES** structure specified in the **DosExecPgm** function call that started the process. If you started the program from the command line, the command processor (*cmd.exe*) makes this value available through the ERRORLEVEL variable. If another process started the program, that process can call the **DosCWait** function to retrieve the error-code value.

If you simply want to exit a given thread, you can call the **DosExit** function with the EXIT_THREAD constant. This call exits the thread without affecting other threads in the process. If the thread hat you exit also happens to be the last thread in the process, the process also exits. If the thread consists of a function, the thread exits when the function returns.

A new thread inherits all the resources currently owned by the process. This means that if you opened a file before creating the thread, that file is available to the thread. Similarly, if the new thread creates or opens a resource, such as another file, that resource is available to the other threads in the process.

A child process inherits resources from the parent process. For example, the child process inherits the current standard input, standard output, and standard error files from the parent process. The child process inherits any resource that was available to the parent process before the parent process created the child process. Once the child process has been created, however, any additional resources that the parent process may create are

not available to the child process. Similarly, any resources that the child process may create are not available to the parent process.

6.6 Terminating a Process

One process can terminate the execution of another process by using the **DosKillProcess** function. The following example terminates the specified process and all child processes belonging to that process:

```
PID pidProcess;

DosKillProcess(DKP_PROCESS, pidProcess);
```

In this example, the pidProcess variable specifies which process to terminate. Typically, you would assign the variable the process identifier for the child process. This is the process identifier that is returned by the **DosExecPgm** function when you started the child process.

6.7 Cleaning Up Before Ending a Process

Since in MS OS/2 any process can terminate any other process (for which it has a process identifier), there is a chance that your program may lose information if a process terminates it before your program can save its work. To prevent this loss of data, you can create a list of exit functions that clean up data and your files before the system terminates the process. The system calls the functions on the list whenever your program is being terminated by another process, or even by itself.

You create an exit list by using the **DosExitList** function. The function takes a function code that specifies an action to take and a pointer to the function that is to receive control on termination. The following example adds the locally defined function SaveFiles to the exit list:

```
#define INCL_SUB
#define INCL_DOSPROCESS
#include <os2.h>

VOID FAR SaveFiles(usTermCode)
USHORT usTermCode;
{
    switch (usTermCode) {
        case TC_EXIT:
        case TC_KILLPROCESS:
             VioWrtTTy("Good-bye\r\n", 10, 0);
             break;
```

```
        case TC_HARDERROR:
        case TC_TRAP:
             break;

    }
    DosExitList(EXLST_EXIT, 0);
}

DosExitList(EXLST_ADD, SaveFiles);
```

Any function that you add to the exit list must be declared with the far attribute and must have one parameter. The function can carry out any task, as just shown, but it must call the **DosExitList** function with the EXLST_EXIT constant as its last action. An exit-list function must not return and must not call the **DosExit** function to terminate.

To execute the exit-list functions, MS OS/2 reassigns thread 1 after terminating any other threads in the process. If thread 1 has already exited (for example, if it called the **DosExit** function without terminating other threads in the process), then the exit-list functions cannot be executed. In general, it is poor practice to terminate thread 1 without terminating all other threads.

You can use the EXLST_REMOVE constant to remove a function from the exit list.

6.8 Creating a Thread

You can use the **DosCreateThread** function to create a new thread for a process. Every process has at least one thread, called the main thread or thread 1. To execute different parts of a program simultaneously, you can start several threads.

To create a thread, you need the address of the code to execute, the address of the first byte in a stack, and a variable to receive the identifier of the thread. The address of the code is typically the address of a function that is defined within the program. The address of the stack can be either an address within a variable declared by using the data segment of the process, or the address in a separate segment. You must not declare the stack within the stack segment of a process. In other words, you must guarantee that the new thread's stack will not be written over by other threads. You must also make sure that there is adequate space on the stack. The amount of space needed depends on a number of factors, including the number of function calls the thread makes and the number of parameters and local variables used by each function. If you plan to call MS OS/2 functions, a reasonable stack size is 4096 bytes.

In the following example, the **DosCreateThread** function creates a thread:

```
USHORT ausStack[2048];
TID tidThread;
VOID FAR ThreadFunc(VOID)
{
    VioWrtTTy("Message from new thread\r\n", 25, 0);
}

DosCreateThread(ThreadFunc, &tidThread, &ausStack[2048]);
```

In this example, the array ausStack is used for the new thread's stack. The thread identifier is copied to the tidThread variable. The thread starts execution with the first statement in the locally defined function ThreadFunc.

In MS OS/2, all stacks grow down in memory; that is, the first byte of the stack is in high memory and the last byte is in low memory, so you need to specify the last word in the stack (in this case, ausStack[2048]) when you supply the starting address of the stack in the call to the function **DosCreateThread**.

A thread continues to run until it calls the **DosExit** function or returns control to the operating system. In the preceding example, the thread exits when the function implicitly returns control at the end of the function.

6.9 Controlling the Execution of a Thread

You can use the **DosSuspendThread** and **DosResumeThread** functions to control the execution of a thread. These functions let you temporarily suspend the execution of a thread if you don't need it and then resume execution when you do need it.

Consider a thread that opens and reads files from the disk. If other threads in the process do not need input from the file, the process can suspend execution of the thread so that the system scheduler does not needlessly grant execution control to the thread.

The **DosSuspendThread** and **DosResumeThread** functions are best used when a process needs to temporarily suspend execution of a thread that is in the middle of a task. For example, a process may suspend a thread that reads files from the disk if the process needs to focus execution on another process.

6.9.1 Suspending a Thread

You can temporarily suspend the execution of a thread for a set interval of time by using the **DosSleep** function. This function suspends execution for the specified number of milliseconds. **DosSleep** is useful if you need to delay the execution of a task. For example, you can use **DosSleep** to delay the response to the user's pressing a DIRECTION key. This gives the user time to observe the results and release the key. The following example uses **DosSleep** to suspend execution of a thread for 1000 milliseconds (1 second):

```
DosSleep(1000L);
```

6.10 Changing the Priority of a Process

You can use the **DosSetPrty** function to change the execution priority of a thread. The execution priority defines when or how often a process receives an execution time slice. Processes with higher priorities receive time slices before those with lower priorities. Processes with equal priority receive time slices in a round-robin order. If you raise the priority of a process, you can guarantee that it executes.

You can set the priority for the entire process, for the process and its child processes, or for just one thread in the process. The following example uses **DosSetPrty** to lower the priority of a process that is intended to be used as a background process:

```
PIDINFO pidiInfo;

DosGetPid(&pidiInfo);
DosSetPrty(PRTYS_PROCESS, PRTYC_IDLETIME, 0, pidiInfo.pid);
```

This example retrieves the process identifier of the current process by using the **DosGetPid** function. It then uses this process identifier to change the priority to an idle-time process (idle-time processes receive the least attention by the scheduler). The **DosGetPid** function retrieves the process and thread identifiers and copies them to the **pid** and **tid** fields in a **PIDINFO** structure.

You can also retrieve the priority of a process by using the **DosGetPrty** function; for example:

```
DosGetPrty(0x0000, &usPriority, pidiInfo.pid);
```

In this example, the function copies the priority of the process specified by pidiInfo.pid to the usPriority variable.

Chapter 7
Lqh: A Sample Program

7.1 Introduction

The **lqh** program is a small-model C-language program that uses MS OS/2 functions and functions of the Microsoft Help library to display the contents of the MS OS/2 System Functions database. Although the **lqh** sample program is similar to the QuickHelp on-line reference program, it is not intended to provide the full functionality of QuickHelp. The sample program described in this chapter shows how to use the MS OS/2 functions to carry out many tasks, such as processing keyboard and mouse input, displaying text and using color on the system screen, and using threads to enhance system performance.

7.2 Lqh Files

The **lqh** program consists of one executable file, *lqh.exe*, and two dynamic-link libraries, *lqhdll.dll* and *mshelp.dll*. The *lqh.exe* file is the **lqh** program file, that is, it contains the program's starting point and most of the functions that process commands. The *lqhdll.dll* file is a dynamic-link library that contains additional functions that **lqh** uses and that may also be useful to other programs. The *mshelp.dll* file is the Microsoft Help library.

The **lqh** program consists of several C-language source files. Most files define the statements of the **lqh** program itself. Others define the statements for the dynamic-link library, *lqhdll.dll*. There are the following source files:

Source Files for lqh.exe

lqh.h
lqhmain.c
lqhinit.c
lqhmenu.c
lqhhelp.c

Source Files for lqhdll.dll

lqhdll.h
box.c
stringsc.c
stringsa.asm

7.3 About the Microsoft Help Library

The **lqh** program relies on the Microsoft Help library to locate and read topics from the MS OS/2 System Functions database. The Help library is a dynamic-link library that provides the functions that are needed to open a database, search for topics, and read topics into memory. Although the sources for the Help library are not provided in the MS OS/2 Programmer's Toolkit, a brief description of each Help function is given in the following list so that you may understand how the library is used to access the database files:

Function	Description
HelpOpen	Opens the database.
HelpClose	Closes the database and frees any memory.
HelpDecomp	Decompresses a topic.
HelpGetCells	Retrieves character cells from the topic.
HelpGetLine	Retrieves a line from the topic.
HelpLook	Searches for a topic.
HelpNc	Retrieves the context number for a topic.
HelpNcBack	Retrieves the previously recorded topic.
HelpNcCb	Retrieves a count of bytes in the topic.
HelpNcNext	Retrieves the next topic.
HelpNcRecord	Records a topic for later display.

7.4 The Lqh Program

The **lqh** program has a simple purpose: open a database to retrieve and display topics to the user. To carry out this purpose, **lqh** uses many MS OS/2 functions from the groups shown in the following list:

- Video (Vio) functions to create a window to view the topic and a menu to show the user commands to choose
- Keyboard (Kbd) and mouse (Mou) functions to let the user choose commands, scroll a topic, and select topics to search for
- Process and semaphore functions to create and manage a second program thread that preloads topics the user may want to view

- Memory-management functions to allocate memory for topics to be loaded

Nearly all of the functions defined in the source files of the **lqh** program use one or more MS OS/2 functions.

7.4.1 The Main Source File

The main source file *lqhmain.c* contains definitions for the following functions:

Function	Description
main	Starts **lqh** and processes commands.
ReadScreen	Reads the word under the cursor from the screen.
ScrollDown	Scrolls the **lqh** window down.
ScrollUp	Scrolls the **lqh** window up.
Abort	Restores the screen and cursor.
MouseRelease	Waits until the mouse button is released.
isalpha	Returns TRUE if the character is a letter of the alphabet.

The main source file includes the **lqh** header file *lqh.h*. This header file contains definitions for the many structures and constants used by **lqh**, as well as function prototypes for the **lqh** functions. The **lqh** header file also includes the MS OS/2 header file *os2.h* and defines the INCL_BASE constant. This constant enables all function, structure, and constant definitions in the MS OS/2 header file.

The main source file defines a variety of global variables. These variables hold values that define the current **lqh** window, the open database, and the state of the mouse and keyboard. For example, the kbdciKeyInfo and mouevEvent structures receive the latest input from the keyboard and mouse, and the selTable array—an array of structures—holds among other things the segment selectors for each decompressed topic in memory. You should take special notice of the hsemNewRef variable. This variable is a RAM semaphore and is used to indicate to the preload thread that references for a new topic should be loaded.

7.4.2 The Initialization Source File

The initialization source file *lqhinit.c* defines the Init function. The Init function opens and initializes a new database. Like other modules, the *lqhinit.c* file includes the **lqh** header file.

The Init function defines several local variables that it uses when opening the database and initializing the selTable structure. The selTable structure contains information about topics that have been preloaded by **lqh**. The function returns TRUE if initialization is successful, or FALSE if the requested database cannot be found or opened.

7.4.3 The Help Source File

The help source file *lqhhelp.c* defines two functions that are used to search for help topics and to preload topics from the database. The HelpSearch function retrieves topics from the current database. The PreLoadThread function runs as a separate thread, preloading topics that are related to the currently displayed topic.

The HelpSearch function retrieves a topic from the database. To identify the topic that is to be retrieved, HelpSearch uses either the null-terminated string pointed to by the first parameter or the value of the third parameter. The second parameter is a Boolean value that specifies whether the HelpSearch function should display a warning message if it does not find the topic. The function returns TRUE if it finds the topic.

The PreLoadThread thread preloads any reference functions that are specified by the current function. First, it waits until the hsemNewRef semaphore is cleared, which indicates that a new function has been loaded. It then goes through the reference table and preloads all the functions. If the hsemNewRef semaphore is set before the thread completes its loop, the thread breaks out of the loop, goes to its beginning, and waits for the semaphore to clear.

7.4.4 The Menu Source File

The menu source file *lqhmenu.c* contains the functions that display the **lqh** menus and carry out actions. It defines the following functions:

Function	Description
Input	Reads keyboard and mouse input.
Reference	Displays the Reference menu and displays the selected topic.
ListMenu	Displays the Categories menu and displays the selected list of topics.
Options	Displays the Options menu and sets the corresponding **lqh** option.
Search	Displays the Search menu and carries out the requested search.

AddBar	Adds a vertical line around the menu name.
DeleteBar	Deletes a vertical line from around the menu name.

7.4.5 The Lqh Make File

To build the **lqh** program, use the **make** program provided with all Microsoft language products, and the **lqh** make file *lqh*. The make file directs **make** to compile and link the **lqh** sources. The file contains the following statements:

```
# Make file for lqh.exe (lqh)
#
# Compiler options:
#   -c: create object file only
#   -Gs  disable stack checking
#     2  create code for 80286
#   -Ox  maximum optimization

CFLAGS = -c -Gs2 -Ox
LIST = lqhmain.obj lqhinit.obj lqhmenu.obj lqhhelp.obj

lqhmenu.obj: lqhmenu.c lqh.h
    cl $(CFLAGS) lqhmenu.c

lqhmain.obj: lqhmain.c lqh.h
    cl $(CFLAGS) lqhmain.c

lqhinit.obj: lqhinit.c lqh.h
    cl $(CFLAGS) lqhinit.c

lqhhelp.obj: lqhhelp.c lqh.h
    cl $(CFLAGS) lqhhelp.c

lqh.exe: $(LIST) lqh.def lqhdll.lib
    link $(LIST), lqh.exe, NUL, lqhdll mshelp /NOI, lqh.def
```

In the command line for the **link** program, the *lqhdll.lib* and *mshelp.lib* files are import libraries for the **lqhdll** and Microsoft Help dynamic-link libraries. These are required to make sure references to functions in the corresponding dynamic-link files are resolved.

7.5 The Lqhdll Dynamic-Link Library

The **lqhdll** dynamic-link library *lqhdll.dll* contains functions that the **lqh** program uses to draw its window, dialog boxes, and message boxes. It also contains functions that copy, concatenate, and compute the length of strings.

The **lqhdll** library contains the following functions:

Function	Purpose
BoxSave	Saves a rectangular area of the screen.
BoxRestore	Restores a rectangular area of the screen.
BoxDraw	Draws a box on the screen.
BoxMessage	Displays a message within a temporary box.
BoxGetString	Displays a box with a prompt and retrieves a string.
BoxScroll	Displays a vertical scroll bar.
Dstrcpy	Copies a string to the given buffer.
Dstrcat	Concatenates a string to the end of another string.
Dstrlen	Returns the length of a string in bytes.

The Box functions provide an easy method for saving and restoring a rectangular area of the screen. The **lqh** program uses the functions to draw the **lqh** window, menus, dialog boxes, and message boxes. For example, it uses the BoxSave function to save the screen along with the current cursor position and cursor type and then uses the BoxDraw function to draw the **lqh** window. As **lqh** is ending, it uses the BoxRestore function to restore the screen, cursor position, and cursor type so that after ending **lqh**, the user can continue working without being forced to redraw the screen.

The **lqh** program uses the string functions in the **lqhdll** dynamic-link library instead of the C run-time functions since the C run-time functions for small-model programs do not support full 32-bit pointers to strings as arguments.

To build the **lqhdll** dynamic-link library, use the **make** program and the **lqhdll** make file *lqhdll*. The make file directs **make** to compile and link the **lqhdll** sources. The file contains the following statements:

```
# Make file for lqhdll.dll (dynamic-link library)
#
#
# Compiler options:
#   -c: Creates object file only
#   -As  small model
#     n  near data pointers
#     u  Upon function entry, DS is saved, then made to point to the named
#        data segment. DS is restored on exit from the function.
#   -Gs  Disables stack checking
#     2  Creates code for 80286
#   -Ox  Optimizes for speed
#   -Zp  Packs structures
#   -Zl  Does not add default libraries
```

```
# Assembler options:
#   /MX  Preserves case in public and external names


CFLAGS = -c -Asnu -Gs2 -Ox -Zp -Zl
ASMFLAGS = /MX
LIST = box.obj stringsc.obj stringsa.obj

box.obj: box.c
    cl $(CFLAGS) box.c

stringsc.obj: stringsc.c
    cl $(CFLAGS) stringsc.c

stringsa.obj: stringsa.asm
    masm $(ASMFLAGS) stringsa.asm;

lqhdll.dll: $(LIST) lqhdll.def
  link  $(LIST), lqhdll.dll, NUL, doscalls /NOD /NOI, lqhdll.def

lqhdll.lib: lqhdll.dll lqhdll.def
    implib lqhdll.lib lqhdll.def
```